

Verifying Complex Systems Probabilistically through Learning, Abstraction and Refinement

Jingyi Wang¹, Sun Jun¹ and Shengchao Qin²

¹Singapore University of Technology and Design ²Teesside University

Abstract. Precisely modeling complex systems like cyber-physical systems is often challenging, which may render model-based system verification techniques like model checking infeasible. To overcome this challenge, we propose a method called LAR to ‘verify’ such complex systems through a combination of learning, abstraction and refinement. Instead of starting with system modeling, our method takes a set of concrete system traces as input. The output is either a counterexample with a bounded probability of being a spurious counterexample, or a probabilistic model based on which the given property is ‘verified’. The model could be viewed as a proof obligation, i.e., the property is verified if the model is correct. It can also be used for subsequent system analysis activities like runtime monitoring. Our method has been implemented as a self-contained software toolkit. The evaluation on multiple benchmark systems as well as a real-world water purification system show promising results.

1 Introduction

Cyber-physical systems (CPS) integrate physical and engineered systems and have the potential to transform the way people interact with engineered systems. They are often used to control public infrastructures like water purification/distribution systems or smart grid systems. When CPS are employed in such safety-critical scenarios, it is desirable to show that they can operate dependably and safely. Analyzing CPS, however, is challenging. Existing system analysis methods, for instance, model-based testing, model checking and theorem proving, require the availability of a system model. Because CPS closely interact with their physical environment, the model should not only capture the system behavior but also the environment’s. Modeling the environment is often hard, due to complicated continuous dynamics in the physical environment.

CPS are merely an example of those complex systems for which manual modeling is challenging. Alternative approaches that do not rely on manual modeling have been explored. One example is statistical model checking (SMC) [40,35]. The idea is to provide a statistical measure on the likelihood of satisfying a property, by observing sample system traces and applying techniques like hypothesis testing [20,40]. However, SMC has its limitations. For instance, since SMC relies on sampling *finite* system executions, it can be challenging to verify un-bounded properties [39,31]. Another approach for avoiding manual modeling is to automatically learn models from sample system traces. Different learning algorithms have been proposed to learn a variety of models, e.g., [34,33,10,16]. It has been shown that such learned models can be useful for system analysis in certain scenarios. Recently, the idea has been extended to learn models

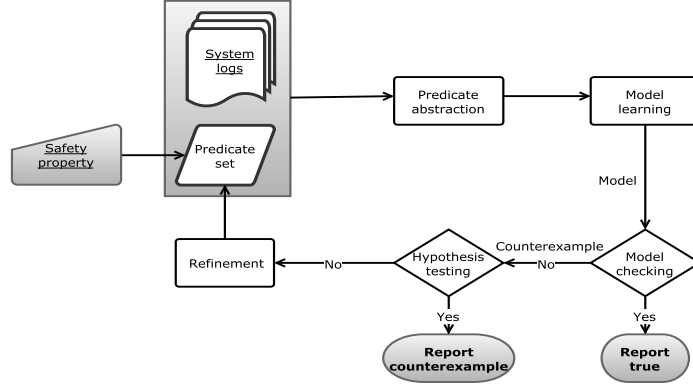


Fig. 1: An overview of our framework

for system verification through model checking. In [28,13,29], the authors proposed to learn probabilistic models and then apply techniques like probabilistic model checking (PMC) to calculate the probability of satisfying a property based on the learned model. Compared to SMC, learning could be beneficial as it solves certain known problems with SMC, e.g., verifying unbounded properties. Furthermore, the learned model could be useful for a range of system analysis or control objectives, e.g., for model-based testing, for implementing runtime monitors, or for designing better system control.

Existing learning methods [34,33,10,16,28,13,29] however have multiple issues. Firstly, they are designed with a fixed level of abstraction, which limits their applicability to real-world systems. For instance, the traces we obtain from a real-world water purification system capture the reading of 50 sensors plus 100 variables used in the control software every 5 milliseconds. Furthermore, these variables are mostly of type float. Without abstraction, it would be hard to learn a reasonably small and precise model of the system. Determining the right level of abstraction is however highly non-trivial. As far as we know, it has not been investigated on how to learn probabilistic models at the right level of abstraction. Secondly, existing learning methods do not take into account the property to be verified. In a recent empirical study [37], it's observed that the verification results based on the learned models could deviate significantly from the actual results. In general, the learned model should be at a level of abstraction which is ideal for verifying or falsifying the property. Furthermore, the verification results based on the learned model must be validated against the actual system.

In this work, we proposed a method named LAR to solve these issues, which works through a combination of probabilistic model learning and counterexample guided abstraction refinement (CEGAR) [14,23]. The overall workflow is shown in Figure 1. The input of LAR includes a safety property and a set of system traces, which could be obtained from a logger in the system. We first construct a set of abstract system traces through predicate abstraction. Next, we apply automatic model learning techniques to construct a probabilistic model of the system, in the form of a discrete time Markov chain (DTMC). Afterwards, we apply PMC to verify the model against the property. If a counterexample is identified, we check whether the counterexample, in the form of a set of paths of the learned model, is spurious. Notice that because we do not have a

model of the system, we cannot completely verify whether the counterexample is spurious or not. Rather, we apply hypothesis testing techniques to bound the probability of it being spurious. If the counterexample is not spurious, we report that the system fails the property. Otherwise, we analyze the counterexample to generate a new predicate which would likely rule out the counterexample. Note that due to the lack of a system model, standard methods for generating new predicates (e.g., weakest pre-condition calculation [14] or interpolation [21]) are infeasible and thus we adopt techniques from the machine learning community to solve the problem. We then repeat the process until either we have identified a counterexample or have constructed a probabilistic model of the system based on which the property is satisfied. We remark that the model generated by our method could be viewed as a proof obligation (i.e., the property is verified if the model is correct), which could be discharged by experts or through other means (like statistical validation of the model). Furthermore, it can be potentially used for subsequent system analysis activities (like for runtime monitoring or better system control). Our method has been implemented as a software toolkit and applied to benchmark systems and a water purification system.

The remainder of the paper is organized as follows. Section 2 defines our problem. Section 3 presents the details of our approach. Section 4 presents our implementation and evaluates our method. Section 5 concludes with a review of related work.

2 Problem Definition

In the following, we review the necessary background and define our problem. For simplicity, we assume the system M under analysis is composed of n observable variables, i.e., $V_M = \{V_1, V_2, \dots, V_m\}$. Hereafter, we omit subscript M if there is no ambiguity. Each variable V_i is associated with a domain D_i . We write $\Sigma(V)$ (hereafter Σ for short) to denote $D_1 \times \dots \times D_m$, which is the set of all possible states of M . Note that Σ may be infinite. We assume that Σ is a countable set. A subset of Σ , denoted as Σ_0 , represents the initial states.

We assume that M is a complicated system such that we do not know exactly how the variables V change. The state of M can be observed through implementing a logger in the system which (e.g., periodically) outputs the valuation of V . Thus, we can obtain a set of finite traces of the system. Formally, a finite system trace is a sequence $\pi = \langle s_1, s_2, \dots, s_n \rangle$ where $s_i \in \Sigma$ for all i . In other words, a trace can be seen as a finite string over Σ . We denote the set of all finite strings over Σ , including the empty string $\langle \rangle$, as Σ^* . Given a string t , a string t' is a prefix of t if and only if there is a string $t'' \in \Sigma^*$ such that $t = t' \cdot t''$ where \cdot denotes string concatenation. We write $prefix(t)$ to denote the set of all prefixes of t .

We assume system M is deterministic given a particular initial state, e.g., M does not contain hidden variables. If we impose a prior probability distribution on the states in Σ_0 (e.g., a uniform distribution over all states in Σ_0), M can be effectively viewed as a discrete time Markov Chain (DTMC). Given a countable set of states S , a probability distribution is a function $\mu : S \rightarrow [0, 1]$ such that $\sum_{s \in S} \mu(s) = 1$. Let $Distr(S)$ be the set of all distributions over S . Formally, M can be viewed as a DTMC (Σ, μ_0, Pr) where Σ is the states of M ; $\mu_0 \in Distr(\Sigma_0)$ is a probability distribution of the ini-

tial states; and $Pr : \Sigma \rightarrow Distr(\Sigma)$ is a transition function such that $Pr(s, s')$ is the probability of transiting from state s to state s' . A DTMC induces an underlying digraph where states act as vertices and there is an edge from s to s' if and only if $Pr(s, s') > 0$. Given a path $\pi = \langle s_1, s_2, \dots, s_n \rangle$ in M , we write $\mathcal{P}(\pi, M) = Pr(s_1, s_2) \times Pr(s_2, s_3) \times \dots \times Pr(s_{n-1}, s_n)$ to denote the probability of exhibiting π in M . Furthermore, we write $Path_{fin}(M)$ to denote the set of finite paths of M starting with an initial state in Σ_0 .

For simplicity, we assume that the property to be verified is specified in a safety fragment of the probabilistic CTL (PCTL) [19,8]. Let AP be a set of atomic propositions constituted by variables in V and let $r \in [0, 1]$ be a real constant. The property is of the form: $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$ where ϕ is a proposition defined on Σ and \mathbf{F} reads ‘eventually’. Intuitively, the property states that the probability of eventually reaching a state satisfying ϕ is no larger than r . For instance, property $\mathcal{P}_{\leq 0.01}(\mathbf{F} \text{waterlevel} \leq 250)$ states that the property of reaching a state with $\text{waterlevel} \leq 250$ is less than 0.01. We remark that our approach can be extended to other safety fragment of PCTL.

Our problem is then to analyze the system in order to answer the following question: given a property $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$, is the property satisfied by M ? We write $M \models \mathcal{P}_{\leq r}(\mathbf{F}\phi)$ to denote that system M satisfies $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$. For instance, in our case study of the water purification system, a safety property is that the water level in the backwash tank must be within certain range. Otherwise, a system shutdown is triggered. Our task is then to show the probability of the water level being out of the range is low enough such that the probability of triggering system shutdown due to that is low.

Since the system is complicated so that a precise model is unknown, it is impossible to formally prove that it satisfies the property. Furthermore, such complex systems are often designed with built-in mechanisms for handling safety violation (e.g., a shutdown sequence is triggered once a violation is detected). The goal is thus to show such safety violation is rare (i.e., with low probability). Because there is no precise model, the outcome of the verification is not simply yes (i.e., satisfied) or no (e.g., by presenting a counterexample). Rather, the outcome should be coupled with some indication on how confident we are on the verification result or what the verification result is based upon.

3 Our Approach

Although the problem is challenging, it is relevant and our approach is an initial attempt towards solving it. Because we cannot precisely model the system, one way to solve the problem is to construct a probabilistic model D (in the form of a DTMC) approximating M and then verify the given property based on D . There are however a number of questions that must be answered in order to make this approach work. First, how do we construct D systematically? In particular, what are the states in D and what is the transition probability? If we consider every different valuation of variables V to be a different state, D is likely to contain many (and often infinitely many) states. We would like to construct a small D (with states more abstract than a valuation of V) which would be used to verify the property. Second, after constructing D and verifying the property based on D , how do we quantify the confidence we have on the verification result, knowing that D may not be precise? In the following, we present our attempt on

answering these questions. We illustrate how our approach works using the following running example (motivated by the *crowds* protocol [30]).

Assume a system where multiple users are browsing the web (i.e., sending/receiving messages to/from Web servers). Further assume there are eavesdroppers on the network who can observe the *direct* source of a message. In order to provide anonymity for the users, a message from a user is not directly sent to the destination, but rather routed among the users so that the eavesdroppers cannot identify the actual source of the message. Each user in the system uses a complicated algorithm to decide on whether to forward a received message to its destination or to some other user on the network. Assume there are a total of u users. The property to be verified is that the probability of observing a message sending to its destination by its actual source more than once in r runs should not exceed a threshold, say 0.2. Formally, it is specified as: $\mathcal{P}_{\leq 0.2}(\mathbf{F}observe0 > 1)$ where *observe0* is a variable in the system which captures the number of message sending to its destination by its actual source. Without knowing how each user decides on forwarding the messages, we cannot develop a precise model.

In our approach, we start with collecting a set of system traces Π , often through introducing a logger in the system and executing the system multiple times. For instance, in the running example, we log the valuation of 33 variables, including where the message is originated, where it is forwarded to, etc. Often, the logged traces contain many details, most of which may not be relevant to verifying the property. Thus, we start with abstracting the traces, through predicate abstraction.

We denote the set of expressions over the set of variables V by $Expr_V$ and the Boolean expressions over V by $BExpr_V$. A proposition φ is a Boolean expression over a set of variables. For an expression $e \in Expr_V$, we denote its valuation in state $s \in \Sigma$ by $\llbracket e \rrbracket_s$. For a Boolean expression e , $\llbracket e \rrbracket_s \in \{0, 1\}$ where 0 stands for *false* and 1 for *true*. We write $s \models \varphi$ iff $\llbracket \varphi \rrbracket_s = 1$. We denote the set of states satisfying a predicate φ by $\llbracket \varphi \rrbracket = \{s \mid s \in \Sigma \wedge s \models \varphi\}$.

Let $P = \{p_1, \dots, p_k\} \subseteq BExpr_V$ be a set of propositions over V . Given a state $s \in \Sigma$, we define an abstraction function as: $\alpha_P(s) = (\llbracket p_1 \rrbracket_s, \dots, \llbracket p_k \rrbracket_s)$, which maps the state s to an abstract state, i.e., a bit vector with length k where each bit represents the truth value of a proposition in P . We write Σ_P to denote the set of abstract states with respects to P . Given a logged trace $\pi = \langle s_1, s_2, \dots, s_n \rangle$, we can construct an abstract trace with respect to predicates P as: $\pi_P = \langle \alpha_P(s_1), \alpha_P(s_2), \dots, \alpha_P(s_n) \rangle$. Let Π be the set of sample traces such that each $\pi \in \Pi$ is a string in Σ^* . We can construct a set of abstract traces Π_P from Π by abstracting each trace in Π one by one.

In our example, we assume that the set Π contains two logged traces: $\langle 0, 0, 1, 1 \rangle$ and $\langle 0, 1, 2, 2 \rangle$ where each number denotes a value of variable *observe0*. Note that we have removed the values of all other variables for simplicity. Based on the above-mentioned property, we set the initial set of predicates P to be $\{observe0 > 1\}$. After predicate abstraction, we obtain two corresponding abstract traces: $\langle 0, 0, 0, 0 \rangle$ and $\langle 0, 0, 1, 1 \rangle$ where a number 0 means that *observe0* ≤ 1 and 1 means *observe0* > 1 .

3.1 Model Learning

With the set of abstract traces Π_P , we then apply existing model learning techniques [28,13,29] to construct a DTMC model D_P . The essential idea of model learning is to construct a

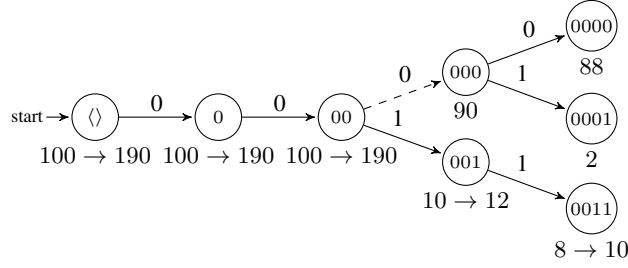


Fig. 2: Example tree representation of samples

DTMC D_P such that it maximizes the probability of observing the traces Π_P as well as contains a relatively small number of states. A number of different learning algorithms have been proposed [34,33,10,16,37]. In the following, we present one of the learning algorithms called AALERGIA [28] as a representative and remark that our approach can be configured to work with different learning algorithms.

The AALERGIA algorithm makes the following assumptions. First, the sample traces are generated by a system which can be modeled as a DTMC, i.e., no non-determinism. Second, the sample traces are mutually independent. Third, the length of each trace is independent from the sequence of states observed. The AALERGIA algorithm has been proved to converge to the actual model if a sufficiently large number of sample traces are provided [28]. However, since the sample traces are often limited in practice, the learned result is not guaranteed to be accurate [37].

Let $\text{prefix}(\Pi_P) = \{\text{prefix}(t) | t \in \Pi_P\}$ be the set of all prefixes of any trace in Π_P . The set of traces Π_P can be naturally organized into a tree $\text{tree}(\Pi_P) = (N, \text{root}, E)$ where each node in N is a member of $\text{prefix}(\Pi_P)$; the root is the empty string $\langle \rangle$; and $E \subseteq N \times N$ is a set of edges such that (π, π') is in E if and only if there exists $e \in \Sigma$ such that $\pi \cdot \langle e \rangle = \pi'$. For instance, assume that Π_P in our running example contains 100 abstract traces: 88 of them are $\langle 0, 0, 0, 0 \rangle$, 2 of them are $\langle 0, 0, 0, 1 \rangle$, 2 of them are $\langle 0, 0, 1 \rangle$, and 8 of them are $\langle 0, 0, 1, 1 \rangle$. Figure 2 shows the tree which represents this set of abstract traces, where each node represents a prefix of some abstract trace in the set. The state labels are explained later.

AALERGIA is inspired by stochastic regular grammatical inference, which aims to learn the structure of a stochastic finite state automaton and estimate its transition probabilities [24]. The idea is to generalize $\text{tree}(\Pi_P)$ by merging the tree nodes according to certain criteria in certain fixed order. Intuitively, two nodes should be merged if they are likely representing the same state in the underlying DTMC. Since by assumption we do not know the underlying DTMC, whether two nodes should be merged is heuristically decided through a *compatibility test*, which intuitively measures how similar two nodes are. We remark the compatibility test effectively controls the degree of generalization. Different types of compatibility test have been studied [10,32,24]. In [10], the compatibility test is based on the Hoeffding bounds, whereas the DSAI algorithm proposed in [16] uses a condition based on calculating certain distance between the two nodes.

In the following, we present the compatibility test adopted in AALERGIA. First, each node π in $\text{tree}(\Pi_P)$ is labeled with the number of traces π' in Π_P such that π is a

prefix of π' . Let $L(\pi)$ denote its label. Two nodes π_1 and π_2 in $tree(\Pi_P)$ are considered compatible if and only if they satisfy two conditions. The first condition is $last(\pi_1) = last(\pi_2)$ where $last(\pi)$ is the last letter in a string π , i.e., π_1 and π_2 must agree on the last abstract state. The second condition is that the future behaviors from π_1 and π_2 must be sufficiently similar (i.e., within Angluin's bound [6]). Formally, given a node π in $tree(\Pi_P)$, we can obtain a probabilistic distribution of the next node by normalizing the labels of the node and its children. In particular, for any event $e \in \Sigma_P$, the probability of going from node π to $\pi \cdot \langle e \rangle$, i.e., the probability of observing e after π , is defined as: $Pr(\pi, \langle e \rangle) = \frac{L(\pi \cdot \langle e \rangle)}{L(\pi)}$. We remark the probability of going from node π to itself, i.e., the probability of not observing the next node, is $Pr(\pi, \langle \rangle) = 1 - \sum_{e \in \Sigma_P} Pr(\pi, \langle e \rangle)$. The multi-step probability from node π to $\pi \cdot \pi'$ where $\pi' = \langle e_1, e_2, \dots, e_k \rangle$, written as $Pr(\pi, \pi')$, is the product of the one-step probabilities.

$$Pr(\pi, \pi') = Pr(\pi, \langle e_1 \rangle) \times Pr(\pi \cdot \langle e_1 \rangle, \langle e_2 \rangle) \times \dots \times Pr(\pi \cdot \langle e_1, e_2, \dots, e_{k-1} \rangle, \langle e_k \rangle)$$

Two nodes π_1 and π_2 are compatible if the following is satisfied: for all $\pi \in \Sigma_P^*$,

$$Pr(\pi_1, \pi) - Pr(\pi_2, \pi) < \sqrt{6\epsilon \log(L(\pi_1))/L(\pi_1)} + \sqrt{6\epsilon \log(L(\pi_2))/L(\pi_2)}$$

Intuitively, it means that the distribution of future traces from π_1 and π_2 must be similar. We highlight that ϵ used in the above condition is a parameter which effectively controls the degree of node merging. Intuitively, a larger ϵ leads to more node merging and subsequently fewer states in the learned model D_P .

If π_1 and π_2 are compatible, the tree is transformed such that the incoming edge of π_2 is directed to π_1 . Next, for any $\pi \in \Sigma_P^*$, $L(\pi_1 \cdot \pi)$ is incremented by $L(\pi_2 \cdot \pi)$. The algorithm works by iteratively identifying nodes which are compatible and merging them until there are no more compatible nodes. The order of choosing merging candidates is hierarchical: first in order of tree depth and for a given depth in the alphabet order of the last observation.

Recall that Figure 2 is the tree representing 100 abstract traces of our running example. The labels on the nodes are the *numbers* of times the corresponding string is a prefix some trace in Π_P . For instance, node $\langle \rangle$ is labeled with 100 because $\langle \rangle$ is the prefix of all traces. This tree can be viewed as the initial learned model which has no generalization. Next, the tree is generalized by merging nodes. Assume that node $\langle 00 \rangle$ and node $\langle 000 \rangle$ in Figure 2 pass the compatibility test so that they are to be merged. First, we update the node label of $\langle 00 \rangle$ to be the sum of the numbers labeling $\langle 00 \rangle$ and $\langle 000 \rangle$. Second, the numbers labeling decedents of $\langle 000 \rangle$ are added to the corresponding decedent nodes of $\langle 00 \rangle$. The result is shown in Figure 2, where the numbers after the arrow are the ones after node merging. For instance, since $L(\langle 000 \rangle \cdot \langle 1 \rangle)$ is 2, we update the label of node $\langle 001 \rangle$ from 10 to 12. Afterwards, the subtree rooted at $\langle 000 \rangle$ (dashed circle nodes, 000 inclusive) is pruned.

After merging all compatible nodes, the last step is to normalize the tree so that it becomes a DTMC D_P . In particular, each node π is taken as a state in D_P . The transition probability from π to a child π' is set to be: $\frac{L(\pi')}{L(\pi)}$ and the probability to itself is set to 1 minus the sum of probabilities to its children accordingly. For instance, in the example shown in Figure 2, the probability transiting from node $\langle 00 \rangle$ to node $\langle 001 \rangle$ is $\frac{12}{190}$, and the probability to itself is $1 - \frac{12}{190}$.

Alternatively, another approach is proposed in [37] to learn probabilistic models based on genetic algorithms (GA later). The idea is to reduce the problem of model learning to an optimization problem, i.e., the problem of finding an optimal DTMC D_P such that the probability of D_P exhibiting Π_P is maximized and the size of D_P is minimized. GA is then applied in the standard way to solve the optimization problem. It has been shown such an approach could outperform AALERGIA sometimes [37]. We remark that different model learning approaches can be adopted in this work.

3.2 Spuriousness Checking

Given the learned DTMC model D_P and the property $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$, we use an existing probabilistic model checker (e.g., PRISM [27]) to check whether $D_P \models \mathcal{P}_{\leq r}(\mathbf{F}\phi)$. We conclude that the original system M satisfies the property if the model checker confirms that $D_P \models \mathcal{P}_{\leq r}(\mathbf{F}\phi)$. However, since this verification result is based on the learned model D_P , which may not be precise according to M , we present the model D_P as a part of the verification result as well, i.e., the property is verified only if D_P is a reasonable model of the system. We remark all model checking results are in fact based on the assumption that the model reflects how the actual system behaves. If the model checker concludes that $D_P \models \mathcal{P}_{\leq r}(\mathbf{F}\phi)$ is not true, we construct a probabilistic counterexample. In general, a probabilistic counterexample for a PCTL property is in the form of a tree [18]. Since we assume that our property is in the form of $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$, we define it to be a set of abstract paths of D_P for simplicity.

Definition 1. *Given a DTMC D_P and a safety PCTL property $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$, a counterexample is a set $C \subseteq \text{Path}_{fin}(D_P)$ such that $\text{last}(\pi) \models \phi$ for every path π in C , and $\sum_{\pi \in C} \mathcal{P}(\pi, D_P) > r$.*

Intuitively, a probabilistic counterexample for $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$ is a set of finite paths in D_P whose accumulated probability is larger than r . There are existing approaches to construct such counterexamples [18]. In our setting, because D_P is learned based on a limited set of abstract traces, a probabilistic counterexample may be spurious. A probabilistic counterexample is spurious if and only if the probability measure of the corresponding concrete paths in M is less than r . Given an abstract path $\pi = \langle s_1, s_2, \dots, s_n \rangle \in \text{Path}_{fin}(D_P)$, we write $\gamma(\pi)$ to denote the set of concrete paths $\{\langle s'_1, s'_2, \dots, s'_n \rangle \in \text{Path}_{fin}(M) \mid \forall i. \alpha_P(s'_i) = s_i\}$ which become π after predicate abstraction using P . Given a probabilistic counterexample C , we write $\gamma(C)$ to denote the set of corresponding concrete paths: $\{\pi' \mid \exists \pi \in C. \pi' \in \gamma(\pi)\}$.

Definition 2. *A set of abstract paths C is a spurious counterexample if and only if $\sum_{\pi \in \gamma(C)} \mathcal{P}(\pi, M) \leq r$.*

We recall that M can be viewed as a DTMC and $\mathcal{P}(\pi, M)$ is the probability of path π in M . The above notion of spurious counterexample is a special case of that of probabilistic automaton defined in [23]. Checking whether a probabilistic counterexample C is spurious or not in our setting is however more challenging than that in the setting of [23]. The reason is that we do not have a model of M and thus the probability of a concrete path in M cannot be calculated. Our remedy is to adopt hypothesis testing [38]

to test whether the hypothesis $\sum_{\pi \in \gamma(C)} \mathcal{P}(\pi, M) > r$ holds given certain error bounds. If it does, we report the counterexample to the user. Otherwise, we conclude that it is a spurious counterexample and proceed to the next step of abstraction refinement.

In the following, we briefly introduce how hypothesis testing is used for spuriousness checking. Hypothesis testing is a statistical process to decide the truthfulness of two mutual exclusive statements. One is H_0 : the hypothesis that the counterexample is not spurious, i.e., $\sum_{\pi \in \gamma(C)} \mathcal{P}(\pi, M) > r$. The other is H_1 : the alternative hypothesis that the counterexample is spurious, i.e., $\sum_{\pi \in \gamma(C)} \mathcal{P}(\pi, M) \leq r$. The probability of making an error is bounded by (α, β) , such that the probability of a Type-I (respectively, a Type-II) error, which accepts H_0 (respectively, H_1) while H_1 (respectively, H_0) holds, is less or equal to α (respectively, β). The test needs to be relaxed with an indifferent region $(r - \delta, r + \delta)$, where neither hypothesis is accepted and the test continues to bound both types of errors [40]. In practice, the error bounds (i.e., α, β), and δ can often be decided by how much testing resources are available. In general, it would require more resource for a smaller error bounds or a smaller indifference region.

Hypothesis testing works by keeping sampling traces from M until a stopping condition is satisfied. There are two main methods to decide when sampling can be stopped. One is fixed-size sampling test, which often results in a large number of tests [41]. The other one is sequential probability ratio test (SPRT), which yields a variable sample size. SPRT is faster than fix-sampling methods as the testing process ends as soon as a conclusion is made. The basic idea of SPRT is to calculate the probability ratio, after observing a trace and evaluate two stopping conditions [7]. If either of the conditions is satisfied, the testing stops and returns which hypothesis is accepted. Readers can refer to [41] for details. We remark that SPRT is guaranteed to terminate [7].

For instance, in our running example, hypothesis testing is to keep generating random messages from some random user in the network and observe the resultant traces. Assume there are now a total of n traces in Π (including the ones obtained initially) and b of them are in $\gamma(C)$. We can then calculate whether to accept H_0 or H_1 based on n and b in the standard way. If neither can be accepted, we continue sampling and updating n and b until one of the hypothesis is accepted.

3.3 Refinement

If hypothesis H_1 is accepted after hypothesis testing, i.e., the counterexample C is spurious, the abstract model D_P is to be refined so as to rule out the spurious counterexample. Following the same idea in CEGAR [14], this is achieved by identifying a new predicate for predicate abstraction. Due to the lack of a system model, standard methods for generating new predicates (e.g., weakest pre-condition calculation [14] or interpolation [21]) are infeasible in our setting. We thus adopt techniques from the machine learning community to solve the problem.

Intuitively, if a probabilistic counterexample C is spurious, there must be at least one path π in C which has certain probability d in D_P , whereas the accumulated probability of $\gamma(\pi)$ in M is less than d . Furthermore, there must be one particular transition in π whose probability in D_P is higher than the accumulated probability of the corresponding concrete transitions in M . The idea is then: if we are able to identify such an

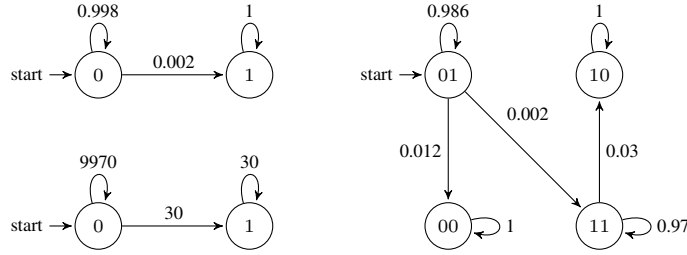


Fig. 3: Identify a spurious transition

abstract path π and such a transition in π , we could refine D_P such that the transition is no longer associated with the overly-high probability so that C may no longer be a counterexample. In the following, we define such transitions and later use them to finding new predicates.

Formally, let $C = \{\pi_1, \pi_2, \dots, \pi_n\}$ be a spurious probabilistic counterexample. Let (s, s') be a pair of consecutive states which form a transition of a path π_i in C . Let $\gamma(s)$ denote the set of concrete states in M which become s after abstraction, i.e., $\gamma(s) = \{x \in \Sigma \mid \llbracket x \rrbracket_P = s\}$, and $|\gamma(s)|$ denote the number of concrete states in $\gamma(s)$. We write $\mathcal{P}(\langle s, s' \rangle, M)$ to be $\sum_{s_0 \in \gamma(s), s'_0 \in \gamma(s')} \mathcal{P}(\langle s_0, s'_0 \rangle, M) / |\gamma(s)|$, i.e., the probability of having a corresponding concrete transition in M . (s, s') is called a spurious transition if $\mathcal{P}(\langle s, s' \rangle, D_P) > \mathcal{P}(\langle s, s' \rangle, M)$. Since we do not have the model M , it is impossible to compute $\mathcal{P}(\langle s, s' \rangle, M)$. Rather, we estimate it based on Π (which now contains all traces sampled during the hypothesis testing in the previous step). That is, given the set of sample traces Π , we estimate $\mathcal{P}(\langle s, s' \rangle, M)$ by $\# \langle s, s' \rangle / \# s$ where $\# \langle s, s' \rangle$ is the number of times the transitions take place and $\# s$ is the number of times s is visited. Next, we identify the most spurious transition, i.e., the transition $\langle s, s' \rangle$ such that $\mathcal{P}(\langle s, s' \rangle, D_P) - \# \langle s, s' \rangle / \# s$ is the largest of all spurious transitions.

Figure 3 shows the first learned model D_P of our running example on the left, where there are two states 0 and 1 representing the state $observe0 \leq 1$ and $observe0 > 1$ respectively. There are in total three transitions: $\langle 0, 0 \rangle$, $\langle 0, 1 \rangle$, $\langle 1, 1 \rangle$. Based on the learned model, $\mathcal{P}(\langle 0, 0 \rangle, D_P)$ is 0.998; $\mathcal{P}(\langle 0, 1 \rangle, D_P)$ is 0.002 and $\mathcal{P}(\langle 1, 1 \rangle, D_P)$ is 1. Assume that the estimation of $\mathcal{P}(\langle 0, 0 \rangle, M)$ is 0.997; the estimation of $\mathcal{P}(\langle 0, 1 \rangle, M)$ is 0.003; and the estimation of $\mathcal{P}(\langle 1, 1 \rangle, M)$ is 1. By calculating the difference between the transition in the learned model and its estimation, we find that the transition $\langle 0, 0 \rangle$ has the largest difference and is thus the most spurious.

Next, we aim to identify a new predicate based on the most spurious transition $\langle s, s' \rangle$. Intuitively, the reason that the transition is spurious is that many abstract paths in D_P going through this transition are infeasible. This is illustrated in Figure 4, where a number of paths are ‘broken’ at the middle state. Note that each abstract state in D_P groups a set of concrete states in Σ . Intuitively, $\langle s, s' \rangle$ is spurious because s groups some states which cannot transit to any concrete states in s' , with states which can. As a result, the probability of transiting from s to s' is inflated. Thus, in order to prune the spurious counterexample, we need to refine the model such that s is split in a way such that the states which cannot transit to any concrete states in s' are separated from the

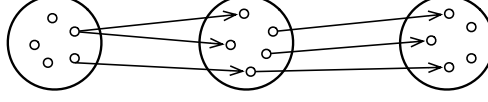


Fig. 4: Illustration of broken paths

rest. In the following, we first collect these two sets of states and identify a predicate for separating them using a classification algorithm.

Given any concrete execution $\pi = \langle s_1, s_2, \dots, s_n \rangle$ in Π , it is clear that we can map any state s_i in the sequence to an abstract state in D_P . Let $\gamma(s, D_P, \Pi)$ be the set of concrete states in any execution in Π which are mapped to state s in D_P . We define $\gamma^+(s, D_P, \Pi) = \{x | x \in \gamma(s, D_P, \Pi) \wedge \exists \langle \dots, x, y, \dots \rangle \in \Pi \wedge y \in \gamma(s', D_P, \Pi)\}$, i.e., the set of concrete states in s which do transit to a concrete state in s' in one of the concrete trace. We define $\gamma^-(s, D_P, \Pi)$ to be $\gamma(s, D_P, \Pi) - \gamma^+(s, D_P, \Pi)$. Figure 5 illustrates how the above works with our running example. Assume that at the top is a concrete trace and at the bottom is the corresponding abstract trace. Given that we identified previously that the spurious transition is $\langle 0, 0 \rangle$, the concrete states s_1, s_2 and s_3 are collected to the set $\gamma^+(s, D_P, \Pi)$ and state s_4 is collected into $\gamma^-(s, D_P, \Pi)$.

Next, we adopt a supervised classification technique to generate a new predicate for refinement. In particular, we use Support Vector Machine (SVM) [15,12] as it is reasonably scalable and produces a predicate as the classifier. SVM is a supervised machine learning algorithm for classification and regression analysis. In this work, we use its binary classification functionality. Mathematically speaking, the binary classification functionality of (linear) SVM works as follows. Given the two sets of concrete states $\gamma^+(s, D_P, \Pi)$ and $\gamma^-(s, D_P, \Pi)$, SVM generates, if there is any, a linear classifier in the form of $e = \sum_i c_i * x_i \geq c$ where $x_i \in V$ is a variable and c_i and c are constant coefficients such that (1) for every state $x \in \gamma^+(s, D_P, \Pi)$, $\llbracket e \rrbracket_x = 1$ and (2) for every state $x \in \gamma^-(s, D_P, \Pi)$, $\llbracket e \rrbracket_x = 0$.

In our running example, since transition $\langle 0, 0 \rangle$ is the most spurious, we collect the two sets of concrete states in order to split state 0 in the DTMC shown on the left of Figure 3. Recall that a concrete state contains the valuation of 33 variables. We feed the two sets into SVM to generate a classifier. The result is a predicate constituted by all 33 variables. To simplify the predicate, we start from the variable with the largest coefficient and try to identify a classifier with a minimum number of variables [15], e.g., a predicate constituted by a minimum number of variables and yet is able to classify the two sets. The result is the predicate $new < runCount$, where $runCount$ is a variable representing how many runs are left and new represents whether it is about to start a new run. Intuitively, this predicate allows us to separate the last run from the other runs, which is relevant because if $observe0$ is 0 after the second last run, it is impossible to reach a state satisfying $observe0 > 1$. The predicate is then added into P , which is then used to refine the abstraction. The right part of Figure 3 shows the learned model after refinement. The new predicate (second bit) introduces a new terminal state 00, which effectively reduces the probability of reaching $observe0 \geq 1$.

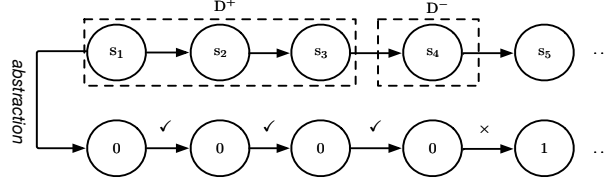


Fig. 5: Labeled data collection at abstract spurious transition $\langle 0, 0 \rangle$

3.4 Overall Algorithm

The overall algorithm is shown as Algorithm 1. The inputs of the algorithms are a set of concrete executions Π , a property in the form of $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$, and the two parameters θ and σ for hypothesis testing. During each iteration of the loop from line 2 to 17, we start with constructing a set of abstract traces based on Π and a set of predicates P . Note that we set the initial set of predicates for abstraction to be the set of propositions in the property. Next, an abstract DTMC D_P is learned using a model learning algorithm. We then verify D_P against the property using PMC. If the property is verified, we conclude that the system is verified and present D_P as a part of the evidence. Otherwise, we construct a probabilistic counterexample C (i.e., a set of abstract paths) at line 9. The spuriousness of C is then statistically checked through hypothesis testing at line 10. If it passes the test, it is returned as a counterexample. In addition to C itself, we output the bounded probability of C being a spurious counterexample. Otherwise, at line 14, we identify the most spurious transition and obtain a new predicate at line 16. After adding the new predicate into P , we restart the process from line 2.

In our running example, after adding the new predicate, we learn the model as shown on the right of Figure 3. Note that state 0 in the model on the left is split into two states 00 and 01. This is due to the new predicate. Note also that the spurious transition $\langle 0, 0 \rangle$ is split into two, which effectively reduces the probability of the counterexample. Verifying this new model against the property using PMC returns success and thus we successfully verify the system without requiring a system model as input.

It is hard to analyze the complexity of this algorithm as it depends on a number of factors, i.e., the initial set of traces P , the complexity of the model learning algorithm, the complexity of hypothesis testing, the complexity of SVM, and whether there is a linear classifier, etc. We thus rely on empirical studies to evaluate the efficiency and effectiveness of the algorithm in the next section. We remark that we do not provide any guarantee on the soundness of the model when a property is verified, though it could be statistically validated through hypothesis testing. Rather it is reported as a part of the verification results which is to be validated separately.

4 Evaluation

Our approach has been implemented as a toolkit named LAR (available at [1]) with about 6K line of Java code. LAR relies on the LIBSVM library [12,4] for generating new predicates and PRISM for PMC [27]. In the following, we evaluate LAR in two

Algorithm 1: Algorithm LAR($\Pi, \mathcal{P}_{\leq r}(\text{trueU}\phi), \alpha, \beta$)

```
1 let  $P$  be the set of atomic propositions in  $\phi$ ;  
2 while true do  
3   construct a set of abstract executions  $\Pi_P$  based on  $\Pi$  and  $P$ ;  
4   apply a model learning algorithm to learn a model  $D_P$  based on  $\Pi_P$ ;  
5   probabilistic model check  $D_P$  against  $\phi$ ;  
6   if  $D_P \models \phi$  then  
7     report  $\phi$  is verified, modulo  $D_P$ ;  
8     return;  
9   construct a probabilistic counterexample  $C$  using the approach in [18];  
10  run hypothesis testing on  $C$  with parameter  $\alpha$  and  $\beta$ ;  
11  if  $C$  is not spurious then  
12    report  $\phi$  is violated and the probability of error is bounded by  $(\alpha, \beta)$ ;  
13    return;  
14  identify the most spurious transitions  $\langle s, s' \rangle$  in  $C$ ;  
15  collect  $\gamma^+(s, D_P, \Pi)$  and  $\gamma^-(s, D_P, \Pi)$ ;  
16  apply SVM to identify a predicate  $p$  separating the two sets;  
17  add  $p$  into  $P$ ;
```

aspects. First, experiments are conducted to see whether LAR could verify properties accurately. Second, how much time is required by LAR. We compare LAR with the approach in [28,37], i.e., to learn a model (using the AALERGIA algorithm and GA respectively) first and then verify the property based on the learned models.

Our test subjects include models from the PRISM benchmark suite [26], as well as a real-world water purification system (SWaT) [3]. For each benchmark in the suite, we use the manually created exact DTMC models of the benchmark systems in [26] to obtain the actual (expected) verification results. To show that LAR is able to learn models providing accurate verification results, the property $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$ is set such that ϕ is the same property for the benchmark in [26] and r is set to be 20% above the actual probability obtained using PRISM. Note that each benchmark model has multiple configurable parameters. SWaT [3] is a complicated system which involves a series of water treatments like ultrafiltration, chemical dosing, dechlorination through an ultra-violet system, etc. We regard SWaT as a representative example of our target systems. Though our approach can be applied to the actual SWaT system, due to safety concerns, we were constrained to conduct our experiment on a simulation software for the system. The software contains an exact Python translation (about 3K LOC) of the control software in SWaT and a set of ODE for simulating the environment. The property we are interested in is how likely the raw water tank in the system would go underflow. All models and detailed results are at [2]. The experiment results are summarized in Table 1, where *TO* means timeout after 2 hours and $-$ means no results. The experiments were conducted in OS X on a machine with 2.6GHz quad-core Intel Core i7 processor, 6M cache and 8 GB RAM.

For each model, a set of traces are first obtained through random simulation. The error bounds for the hypothesis testing are set as: $\alpha = 0.01, \beta = 0.01$. The first four

Case study (parameters)	ap	states	expected	AA			GA			LAR			
				states	time	result	states	time	result	states	iterations	time	result
<i>crowds</i> (R,S)	5,5	8k	0.1458	8k	767	0.1794	100	665	0.1658	4	2	<u>37</u>	0.1794
	5,10	p1 111k	0.1048	10k	1953	0.1054	158	472	0.1054	4	2	<u>36</u>	0.1054
	5,15	p1 592k	0.0922	–	TO	–	205	578	0.0714	4	2	<u>37</u>	0.0714
	5,20	2062k	0.0861	–	TO	–	263	742	0.0827	4	2	<u>36</u>	0.0827
<i>egl</i> (L,N)	2,5	p1 29k	0.5156	–	TO	–	–	TO	–	4	1	<u>6</u>	0.5039
		p2	0.4844										0.4961
	2,10	p1 6E5k	0.5005	–	TO	–	–	TO	–	4	1	<u>21</u>	0.5357
		p2	0.4995										0.4643
<i>nand</i> (N,K)	20,2	155k	0.4129	2k	166	0.5657	89	323	0.54	9	5	<u>149</u>	0.54
	20,3	232k	0.4685	2k	240	0.4569	90	401	0.4999	9	5	<u>125</u>	0.4972
	40,2	2003k	0.4838	7k	1726	0.52	171	832	0.5199	20	11	<u>721</u>	0.48
	40,3	p1 3001k	0.5777	6k	<u>839</u>	0.8331	168	972	0.8331	29	11	1322	0.7777
	60,1	4717k	0.2695	12k	3320	0.4285	247	1145	0.4285	24	8	<u>481</u>	0.1786
	60,3	1.4E5k	0.6377	8k	1451	–	8k	1278	–	17	7	<u>598</u>	0.75
<i>swat</i> (S,R)	5,1	p1 ∞	0.1713	20k	TO	–	20k	TO	–	134	9	<u>1629</u>	0.1822
	5,5	p1 ∞	0.1389	20k	TO	–	20k	TO	–	126	3	<u>332</u>	0.1428
	10,5	p1 ∞	0.3333	20k	TO	–	20k	TO	–	535	8	<u>664</u>	0.2355

Table 1: Experiment results: ‘ap’ is the proposition ϕ in the property; ‘expected’ is the actual probability of satisfying ϕ ; ‘iterations’ is the number of iterations for LAR.

columns of Table 1 show the parameters of the test subjects, the properties to verify and the size of the state space. The following column shows the expected verification result. The next six columns show the statistics of the approach in [28] (named AA for simplicity) and GA in [37] respectively, i.e., the number of states in the learned models, the time cost in seconds and the verification result based on the learned models. The last four column shows the statistics of LAR. The column ‘iterations’ is the number of loops in Algorithm 1. We remark that although the answer to the properties (i.e., in the form of $\mathcal{P}_{\leq r}(\mathbf{F}\phi)$) is yes or no, we show the probability of satisfying property ϕ obtained through PMC based on the learned model so that we can observe how accurate the learned models are with respect to the property.

We have the following observations based on the results in Table 1. First, the models generated by LAR (as a part of the evidence why the property is claimed verified) have much fewer states than AA or GA, i.e., often orders of magnitudes fewer than the model learned by AA. The reason is that we always start with learning based on the most abstract traces and add 1 predicate for abstraction every iteration. In most cases, LAR only takes a few iterations to identify a model which allows us to prove the property. This is particularly useful in verifying systems with a large number of states. For instance, for the *egl* model, while AA (or GA) fails to learn or verify any of 4 cases, LAR successfully verifies all of them in less than 30 seconds and the learned models have 4 states only. Second, LAR is often much faster than AA and GA. This is correlated to the fact that a much smaller DTMC is learned, since the abstract traces have only a few states (up to $2^{\#P}$ where $\#P$ is the number of predicates in P). Furthermore, because the learned model is small, checking whether the model satisfies the property through PMC takes little time. A closer look reveals that spuriousness checking through

hypothesis testing dominates LAR’s time. We acknowledge that if multiple iterations of abstract refinement is necessary, LAR may take a relatively long time, as observed in the case of the *nand* models. Third, the models produced by LAR often (i.e., in 15 out of 17 cases) have more precise verification results than those learned by AA and GA. This is particularly obvious for the SWaT system. The proposition ϕ here is $waterlevel \leq 250$. Because the variables in this system are all of float type, without abstraction, every logged state is considered as a different one and as a result, there is no generalization (i.e., no state merging in AA and GA) and as a consequence there are lots of time-consuming compatibility tests. As a result, both AA and GA fail to terminate within timeout. On the other hand, LAR is able to learn reasonably small models and relatively accurate results compared to the estimated probability.

5 Conclusion and Related Work

This work is our initial attempt on solving the problem of verifying complicated systems without manual modeling. The proposed method is to verify complex systems probabilistically through a combination of learning, abstraction and refinement. The preliminary experimental results show promising results, i.e., not only LAR verifies systems efficiently but also LAR generates models which could be useful for other purposes.

This work is inspired by the recent work on learning models for model checking. In order to avoid manual modeling, machine learning has recently been adopted to learn a variety of system models (e.g., DTMC, CTMC, stationary models and MDPs) for model checking [28,34,13,29]. Existing learning algorithms are often based on algorithms designed for learning (probabilistic) automata, as evidenced in [33,32,10,16,11,5]. In addition, in [9], reinforcement learning algorithms are applied in order to verify Markov decision processes, without constructing explicit models. LAR complements these model learning approaches with a framework similar to CEGAR so that model learning can be done in a more meaningful way (i.e., learn at the right level-of-abstraction and learn towards the verification of certain property).

This work is also related to work on SMC. Similarly to our work, SMC [40,35] can be applied when the system model is not available. Hypothesis testing is initially adopted by SMC mainly for bounded properties. There are some recent work on extending SMC to unbounded properties [39,31]. The main difference between LAR and SMC is that LAR generates models as a part of the verification results, which would offer insights on how the system works and why the property is verified. Furthermore, because LAR verifies the system based on the learned model, it is not limited to bounded properties.

This work is built upon the work on CEGAR and in particular the work on extending CEGAR to probabilistic systems. CEGAR was proposed in [14] and developed further later [22,21,25]. The fundamental questions and pragmatic issues of probabilistic abstraction refinement are first explored in [23] in the context of predicate abstraction [17,36]. In particular, the authors of [23] proposed probabilistic CEGAR, where probabilistic counterexample (obtained using the approach documented in [18]) is used to refine an abstraction. Compared to the CEGAR and probabilistic CEGAR, LAR is different as LAR does not require any user-provided system model.

References

1. https://jingyi_wang@bitbucket.org/jingyi_wang/ziqian_develop.
2. https://jingyi_wang@bitbucket.org/jingyi_wang/ziqian_evaluation.
3. <http://itrust.sutd.edu.sg/research/testbeds/secure-water-treatment-swat/>, 2015.
4. Thomas Abeel, Yves Van de Peer, and Yvan Saeys. Java-ml: A machine learning library. *The Journal of Machine Learning Research*, 10:931–934, 2009.
5. Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.
6. Dana Angluin. Identifying languages from stochastic examples. 1988.
7. A. Wald. *Sequential Analysis*. Wiley, 1947.
8. Andrea Bianco and Luca De Alfaro. Model checking of probabilistic and nondeterministic systems. In *Foundations of Software Technology and Theoretical Computer Science*, pages 499–513. Springer, 1995.
9. Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelík, Vojtěch Forejt, Jan Křetínský, Marta Kwiatkowska, David Parker, and Mateusz Ujma. Verification of markov decision processes using learning algorithms. In *Automated Technology for Verification and Analysis*, pages 98–114. Springer, 2014.
10. Rafael C Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In *Grammatical Inference and Applications*, pages 139–152. Springer, 1994.
11. Rafael C Carrasco and Jose Oncina. Learning deterministic regular grammars from stochastic samples in polynomial time. *Informatique théorique et applications*, 33(1):1–19, 1999.
12. Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.
13. Yingke Chen, Hua Mao, Manfred Jaeger, Thomas Dyhre Nielsen, Kim Guldstrand Larsen, and Brian Nielsen. Learning markov models for stationary system behaviors. In *NASA Formal Methods*, pages 216–230. Springer, 2012.
14. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.
15. Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
16. Colin De la Higuera. *Grammatical inference*, volume 96. Cambridge University Press Cambridge, 2010.
17. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Computer aided verification*, pages 72–83. Springer, 1997.
18. Tingting Han and Joost-Pieter Katoen. *Counterexamples in probabilistic model checking*. Springer, 2007.
19. Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, 6(5):512–535, 1994.
20. Klaus Havelund and Grigore Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 342–356. Springer, 2002.
21. Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L McMillan. Abstractions from proofs. In *ACM SIGPLAN Notices*, volume 39, pages 232–244. ACM, 2004.
22. Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. *ACM SIGPLAN Notices*, 37(1):58–70, 2002.

23. Holger Hermanns, Björn Wachter, and Lijun Zhang. Probabilistic cegar. In *Computer Aided Verification*, pages 162–175. Springer, 2008.
24. Christopher Kermorvant and Pierre Dupont. Stochastic grammatical inference with multinomial tests. In *Grammatical Inference: Algorithms and Applications*, pages 149–160. Springer, 2002.
25. Daniel Kroening, Alex Groce, and Edmund Clarke. Counterexample guided abstraction refinement via program execution. In *Formal Methods and Software Engineering*, pages 224–238. Springer, 2004.
26. M. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. 9th International Conference on Quantitative Evaluation of Systems (QEST'12)*, pages 203–204. IEEE CS Press, 2012.
27. Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.
28. Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D Nielsen, Kim G Larsen, and Brian Nielsen. Learning probabilistic automata for model checking. In *Quantitative Evaluation of Systems (QEST), 2011 Eighth International Conference on*, pages 111–120. IEEE, 2011.
29. Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D Nielsen, Kim G Larsen, and Brian Nielsen. Learning markov decision processes for model checking. *arXiv preprint arXiv:1212.3873*, 2012.
30. M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92, 1998.
31. Christian Rohr. Simulative model checking of steady state and time-unbounded temporal operators. In *Transactions on Petri Nets and Other Models of Concurrency VIII*, pages 142–158. Springer, 2013.
32. Dana Ron, Yoram Singer, and Naftali Tishby. On the learnability and usage of acyclic probabilistic finite automata. In *Proceedings of the eighth annual conference on Computational learning theory*, pages 31–40. ACM, 1995.
33. Dana Ron, Yoram Singer, and Naftali Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine learning*, 25(2-3):117–149, 1996.
34. Koushik Sen, Mahesh Viswanathan, and Gul Agha. Learning continuous time markov chains from sample executions. In *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, pages 146–155. IEEE, 2004.
35. Koushik Sen, Mahesh Viswanathan, and Gul Agha. Statistical model checking of black-box probabilistic systems. In *Computer Aided Verification*, pages 202–215. Springer, 2004.
36. Bjorn Wachter, Lijun Zhang, and Holger Hermanns. Probabilistic model checking modulo theories. In *null*, pages 129–140. IEEE, 2007.
37. Jingyi Wang, Jun Sun, Qixia Yuan, and Jun Pang. Should we learn probabilistic models for model checking? a new approach and an empirical study, 2016.
38. Hakan L Younes. Verification and planning for stochastic processes with asynchronous events. Technical report, DTIC Document, 2005.
39. Håkan LS Younes, Edmund M Clarke, and Paolo Zuliani. Statistical verification of probabilistic properties with unbounded until. In *Formal Methods: Foundations and Applications*, pages 144–160. Springer, 2011.
40. Håkan LS Younes and Reid G Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In *Computer Aided Verification*, pages 223–235. Springer, 2002.
41. H.L.S Younes. *Verification and Planning for Stochastic Processes with Asynchronous Events*. PhD thesis, Carnegie Mellon, 2005.